CALIFORNIA INSTITUTE OF TECHNOLOGY

Computer Science

5040:TR:82

CONCURRENT ALGORITHMS FOR THE CONJUGATE GRADIENT METHOD

by

Lennart Johnsson

Copyright California Institute of Technology, 1982

Table of Contents

List of Figures

List of Tables

CONCURRENT ALGORITHMS FOR THE CONJUGATE GRADIENT METHOD

Lennart Johnsson

Computer Science
California Institute of Technology
Pasadena, CA. 91125

ABSTRACT

A few concurrent algorithms for the basic conjugate gradient method is devised and discussed. Most of the algorithms have a topology that is naturally determined by characteristic dimensions of the system and the operations of each step of the conjugate gradient method. The topologies map well onto buildable structures of sparsely interconnected processors while preserving unit communication distance. The topology of the algorithms are:

- 1) A binary tree
- 2) A composition of a binary tree and a ring the nodes of which forms the leaves of the tree.
- 3) A linear array with some additional processing elements.

It is also discussed how these algorithms maps onto Boolean n-cubes.

The algorithms all have the property that a communication operation is associated with each computation.

No claim is made as to the optimality from a space-time complexity point of the algorithms presented here. However, the processor utilization for some algorithms and topologies are close to 100% and the space*time complexity of those algorithms are of the same order as the arithmetic complexity of common sequential machine algorithms.

1 INTRODUCTION

The conjugate gradient method (CGM) is an exact method for solving linear systems of equations. The solution is obtained in N steps, where N is the dimension of the system. There exist several variations on the basic method by [Hestenes, Stiefel 52]. The variations are typically made to enhance convergence, reduce storage requirements, extend the applicability of the method (e.g., to indefinite systems), or some combination thereof.

A particular class of modifications of the basic CGM is often called preconditioned CGM, see e.g. [Meijerink, van der Vorst 77], [Meijerink, van der Vorst 78], [Kershaw 78], [Munksgaard 79], [Jennigs, Malik 78], [Manteuffel 80], [Glowinski et. al. 80], [van der Vorst 82]. Preconditioned CGM will be treated in a separate document. Biorthogonalization methods, such as Lanczos' method are computationally very similar to the basic CGM. A number of different biorthogonalization methods are described in [Lam 76], [Saad 80], and [Saad 81].

Here a number of concurrent algorithms for the basic conjugate gradient method is devised and discussed. The algorithms are of different space and time complexity. The linear system of equations is assumed to have N unkowns and N equations. The concurrency in several of the algorithms are related to the dimension N or for a banded matrix the bandwidth b.

Sparsity is only considered in terms of band structure.

In the following only the computations in the iteration sequence is considered. Loading and unloading of data is disregarded.

Some parts of the algorithms are described using a graphic notation instead of a progamming or mathematical notation. The graphic notation might better illuminate the complexity and topology of a direct hardware implementation whenever that is being considered.

A crucial part of the conjugate gradient method is the computation of inner products. A substantial part of this report is devoted to this subproblem.

The conjugate gradient method (CGM) can be defined as follows:

$$a_i = (r_i, r_i)/(p_i, Ap_i)$$

$$x_{i+1} = x_i + a_i p_i$$

$$r_{i+1} = r_i - a_i A p_i$$

$$b_i = (r_{i+1}, r_{i+1})/(r_i, r_i)$$

$$p_{i+1} = r_{i+1} + b_i p_i$$

The dimension of the vectors is N, and the matrix A is N by N.

Each step of the algorithm includes vector additions/subtractions, computation of inner products, matrix-vector products, and scalar operations. The time required for these operations depends on the number and the capabilities of processors available, their interconnection, and the distribution of data among the processors.

The scalar a_i has to be computed before any of the components of x_{i+1} and r_{i+1} can be computed. Similarly b_i has to be known before any of the components of p_{i+1} can be computed. The coefficient a_i can be computed in $O(\log_2 N)$ time steps using a tree structure, or in $O(N)$ time using a linear array. The other computations associated with one step of the algorithm can be carried out in one or a few time steps, independent of N. Hence, the time required by the CGM for each step is $O(\log_2 N)$ to $O(N)$. The number of steps to compute the solution of a linear system of equations is N.

The computations of succesive steps of the algorithm can not be pipelined to make the CGM linear in time. The problems stem from the computation of the coefficients a_i and b_i, which determines how far in the direction of the vector p_i each step of the algorithm shall proceed. The determination of this distance is based on the norm of the residual vector and the quadratic form (p_i, Ap_i). These quantities are global and enforces a dependence between the steps in the different dimensions of the space. Communication between all N dimensions is required for each step of the algorithm.

Biorthogonalization methods have the same limitation from a concurrency point of wiev. The algorithms discussed in this report for the basic CGM can be extended in a fairly straightforward way to become applicable for the biorthogonalization methods.

Decoupling the iterations in the different dimensions of the space is desirable with respect to concurrency. However, the convergence rate is likely to be decreased by such a basic change in the algorithm.

We will now consider the "macro" operations of the CGM one by one, and then all of them together. Concurrent algorithms of different time-space complexities will be presented. The mapping of an in-space fully instantiated algorithm onto an array of fewer processing elements than required by such an algorithm will also be discussed.

In what follows it is assumed that a time step is defined by the time required to perform a multiply/add operation including one communication. This is a fairly coarse subdivision of time. By using a finer subdivision some increase in performance can be gained by pipelining and interleaving operations at lower levels than discussed here. However, the primary goal here is to structure data and algorithms in such a way that an increase in performance can be achieved by having several processors actively participate in the computational task. With suitable algorithms and a large number of processors available, the time required to compute a solution can be reduced by orders of magnitude.

2 "MACRO" OPERATIONS

Vector addition can obviously be carried out in one time step if the corresponding components of the two vectors are in the same processor, and there are N processors each of which holds one set of components.

Multiplication of a vector by a scalar can also be computed in one time step if each product is formed in separate processors and the scalar is either available in each of the processors or broadcasted to all processors.

An inner product between two vectors can be computed in:

- $\log_2 N$ time steps by having N processors for the multiplications and performing the additions in a binary tree of adders, given that the data are properly distributed between the processors. Successive inner products can be computed every cycle by employing pipelining techniques.

- N time steps by one multiply/add processor and 2N storage cells for the vectors. If several vector products are to be computed the time for computing all products obviously need not increase if sufficiently many processors are available to allow all vector products to be computed concurrently.

A matrix-vector product is equivalent to computing N inner products. A matrix-vector product can be computed in:

- $\log_2 N$ time if N trees as discussed above are available and the vector can be broadcasted to the trees. The components of the product vector is available at the roots of the trees, one component in each root. The number of processors used are $2N^2 - N$.

- $N + \log_2 N$ time if the computations are pipelined through one tree of processors. The components of the product vector are available one by one at the root of the tree. 2N-1 processors are used in the tree.

- 2N time if N processors are used.

2.1 Band matrices

The discussion above paid no particular attention to the structure of the matrix A. A particularly simple structure is defined by a band matrix. For such a matrix algorithms of a lower space-time complexity than full matrix algorithms can be devised.

A few examples of computational networks for computing a matrix-vector products is presented next. The matrix is assumed to be

of band type with r codiagonals below and s codiagonals above the main diagonal, and bandwidth b = r+s+1.

2.1.1 Computation of Matrix-vector products, Ax

Alternative 1, N(2b-1) processors, $\log_2 b$ time steps

In a direct full instantiation in space all Nb products can be formed in one step. Employing a tree of adders for each row allows for the summation to be made in $\log_2 b$ steps. The components of the product vector are all available in the same time step.

Alternative 2, N processors, b time steps

A) In this alternative a processor is associated with each row of the matrix.

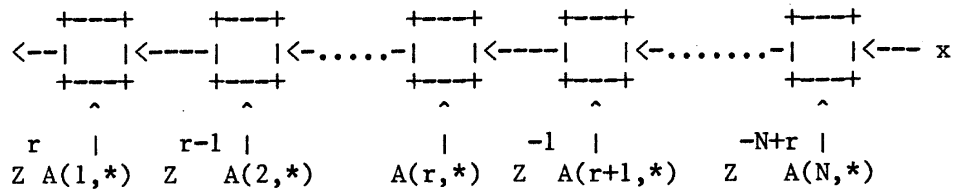

Figure 1: A row organized array for matrix-vector products

A(i,*) denotes row i of the matrix A. The elements of a row are assumed to enter in column order. The Z denotes a delay of one time step. With the delays in Figure 1 codiagonals are entered concurrently if the array is operated in a synchronous manner. The function of each box in Figure 1 is described by Figure 2.

The set of computations that occurs concurrently in the array in Figure 1 can be illustrated by wave fronts which, again assuming synchronous operation, takes the form shown in Figure 3.

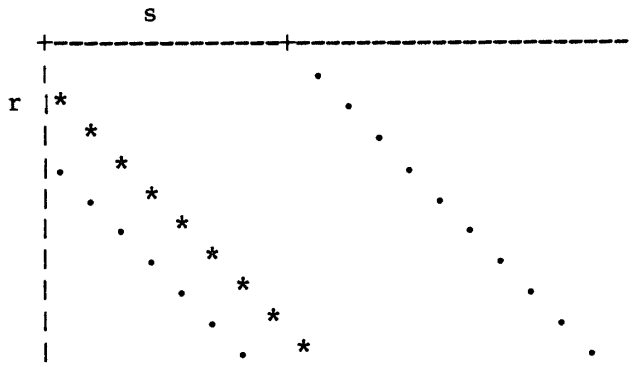

Figure 3: Wave fronts of the matrix-vector computation in Figure 1

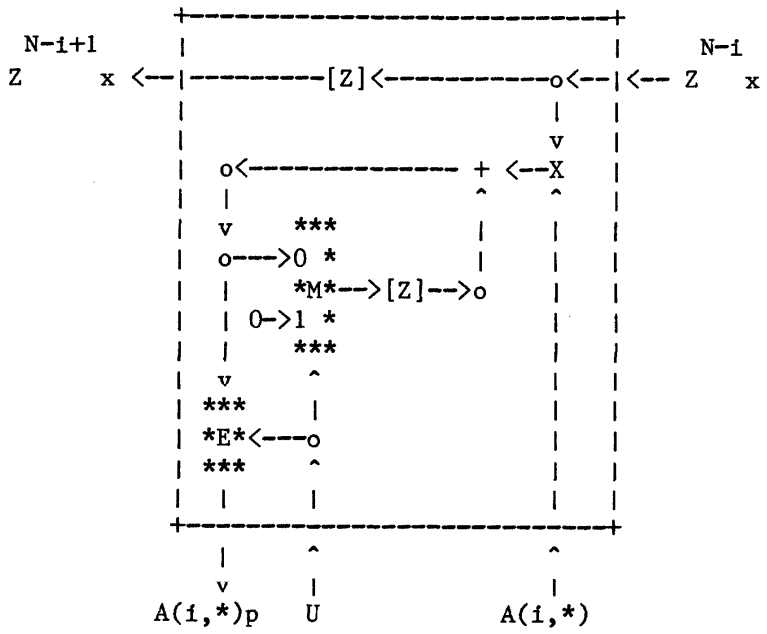

Figure 2: A row processor for matrix-vector products

The wave front is indicated by asterisks in Figure 3. It moves from the left side of the band to the right side, given that the rows of A are furnished in column order and the elements of x in the corresponding order. It is obviously also possible to supply the data in the reversed order whereby the wavefronts will move in the opposite direction.

This alternative computes the matrix-vector product in b time steps using N processors. All components of the matrix-vector product are available during the same time step, i.e., the output is parallel. In order for this alternative not to be performance limited by its input, the first N-r components of x has to be loaded in parallel.

B) Instead of having a processor for each row of the matrix it is also possible to associate a processor with each column. The vector will be distributed with one component per processor. Partial product sums (PPS) have to be passed between processors. A PPS moves b processors from its origin in order to accumulate all its product terms.

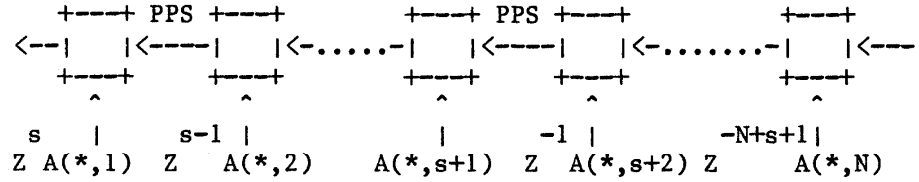

Figure 4: A column organized array for matrix-vector products

The functions of a processor are illustrated graphically in Figure 5.

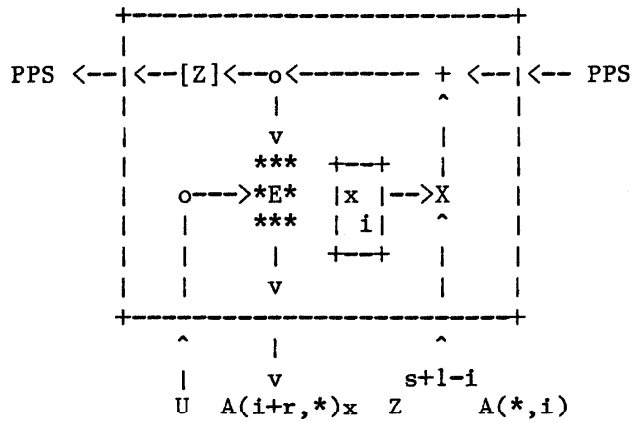

Figure 5: A column processor for matrix-vector products

The first r components of the product will appear sequentially from unit number 1 during the steps s+1 to s+r+1. The remaining components will be available during step s+r+1.

The wave fronts are as in Figure 3. With each column being supplied in row order the wave fronts move from the upper part of the band to the lower part.

It should be noticed that a major difference between alternatives A and B is that in the latter b units are required for every component of the product, except for the first and last few components.

The product is again computed in b steps by N processing elements.

C) It is also possible to have partial product sums moving in one direction and the vector in the opposite direction. The data supplied to a unit in this case form antidiagonals. Hence, 2N-1 units are required to compute the product in b steps. The components of the vector are at any time located in every other processor. The direction of information flow is shown in Figure 6, and the functionality of the processing units is shown in Figure 7. The wave fronts are again as in Figure 3.

Alternative C is less efficient and has a more complex behavior than alternatives A and B.

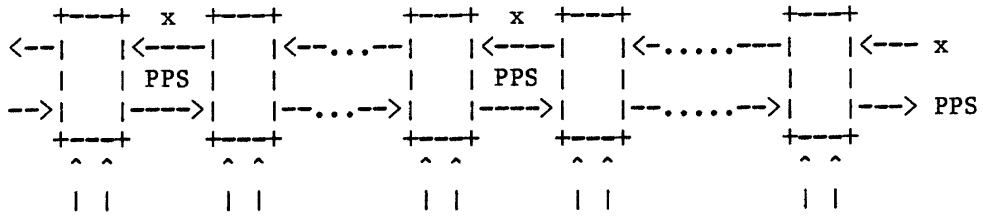

Figure 6: An antidiagonally organized array for matrix-vector products

Remark: The wave fronts of all three alternatives above are the same

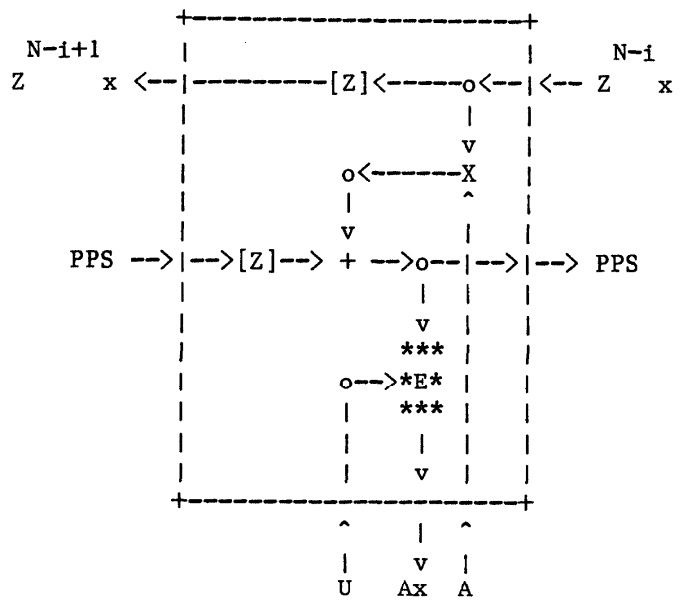

Figure 7: An antidiagonal processor for matrix-vector products

even though the data streams consist of different data sets. All components of the matrix-vector product are available during the last time step.

Alternative 3, b processors, N time

A) In alternative 2A one of the inputs to each processor is a row of the matrix. Neighboring rows are skewed one step with respect to each other. The same organization and processing units can also be used for the case where the number of processors is b. In this case b consecutive rows are input concurrently and contiguous sets of b rows are being input sequentially. Sets of b consecutive components of the product are available during every b cycle. The input streams containing the matrix A are illustrated in Figure 8.

The time to compute the matrix-vector product is N+b disregarding "edge" effects.

Sequential output can be obtained without changing the function of the processing elements by starting the computations in neighboring processors at successive instances of time. Due to the interlocking between the processors caused by the shifting of the vector x through the array, the vector components and hence also the matrix coefficients have to be at a distance of two apart. The computation time is essentially doubled to 2(N+b).

B) Broadcasting is a facility that often may be undesirable, particularly in VLSI technology. However, if sequential output is desirable it can be obtained at no additional time steps using broadcasting. Broadcasting of the components of the vector is not

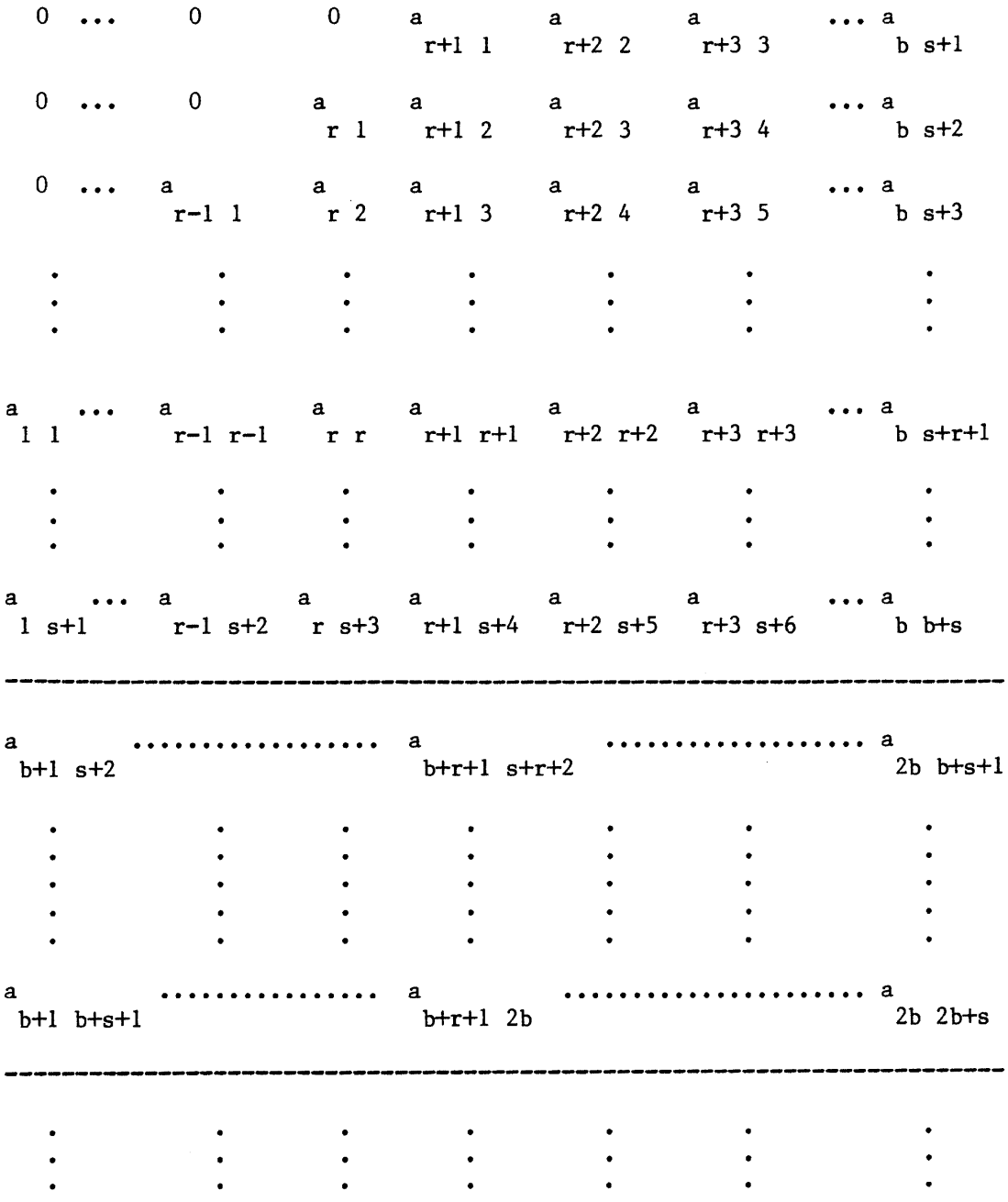

Figure 8: The matrix A formed into the data streams of 3A

useful in alternative 2 since only b matrix coefficients use the same component of the vector.

The function of the processing units is illustrated by Figure 9 (compare Figure 2).

The wave fronts take a different form when broadcasting of the vector is used, Figure 10.

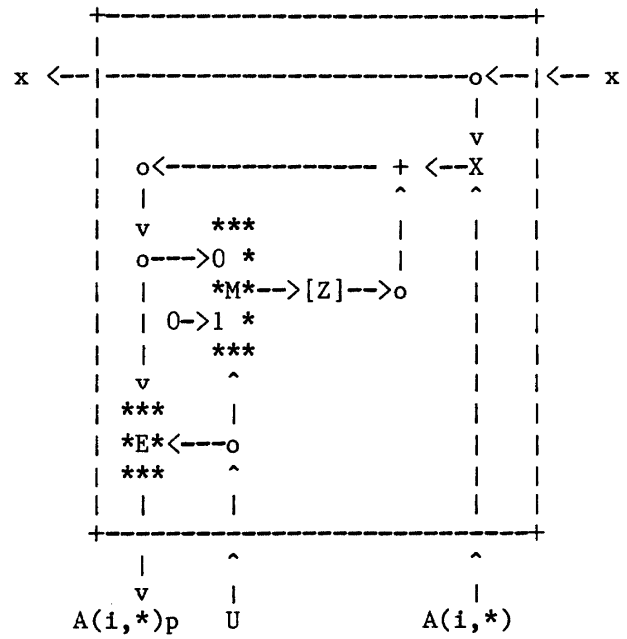

Figure 9: A row processor with broadcasting

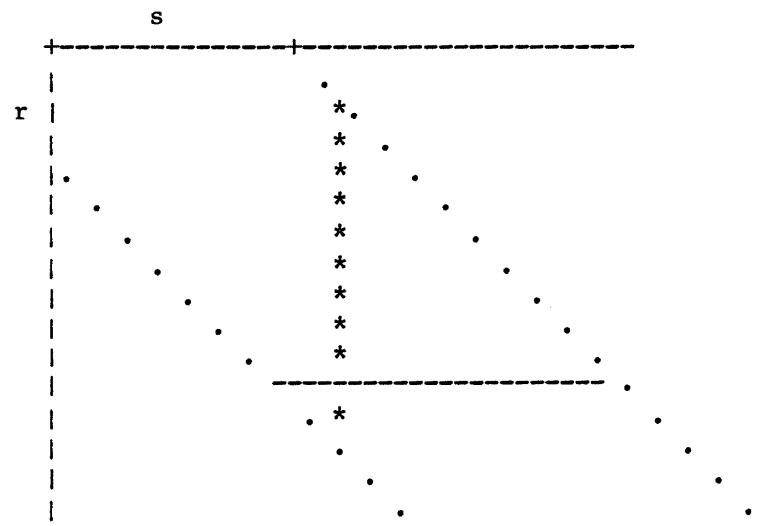

Figure 10: Wave fronts in matrix-vector computation with broadcasting of the vector

C) As in alternative 2 it is clearly also possible to organize the computations columnwise. With the vector components stationary as in alternative 2B and the column computations skewed with respect to each other in such a way that the computations in a column, i+1, is one row ahead of the computations in column i, the wave fronts are as in Figure 3. The partial product sums move from column i+1 to column i.

It is also possible to let the computations of column i precede the

computations of column i+1 by one row. In this case the partial product sums will instead be moving from column i to column i+1. The wave fronts will in this case correspond to antidiagonals of a matrix that is N by b. Sets of b columns are entering the array in sequence. The set of active computations at any given time are shown as wave fronts in Figure 11. If the band matrix is rearranged into a N by b matrix then the wave fronts fall on one "antidiagonal".

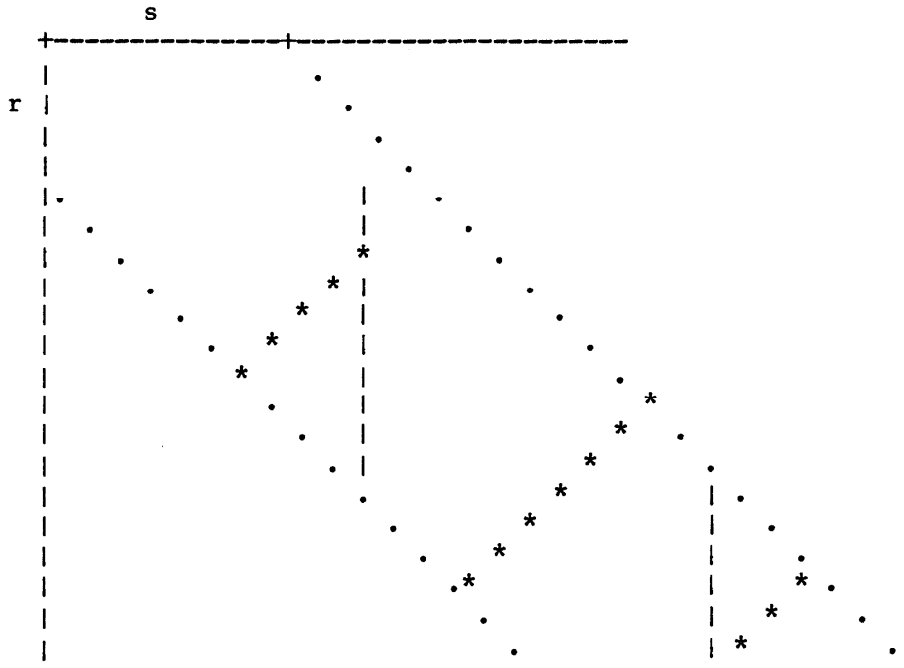

Figure 11: Wave fronts for one kind of columnwise matrix-vector product computation

The output will be sequential, except possibly for the last few components of the product.

It should be noticed that with the columnwise arrangement only partial product sums are formed in the computations associated with each set of b columns. However, there is no need to explicitly save and reload the partial product sums, since they are in the correct locations for continuing the computations with the next set of b columns in order of increasing column index.

The b active components of the vector x have to be loaded in parallel every b time steps.

Disregarding edge effects the time to compute the product is N+b.

The organization of the matrix data is as in Figure 12.

D) There is also a natural correspondent to alternative 2C. This alternative is described in [Kung, Leiserson 80] and is included here for the sake of completeness.

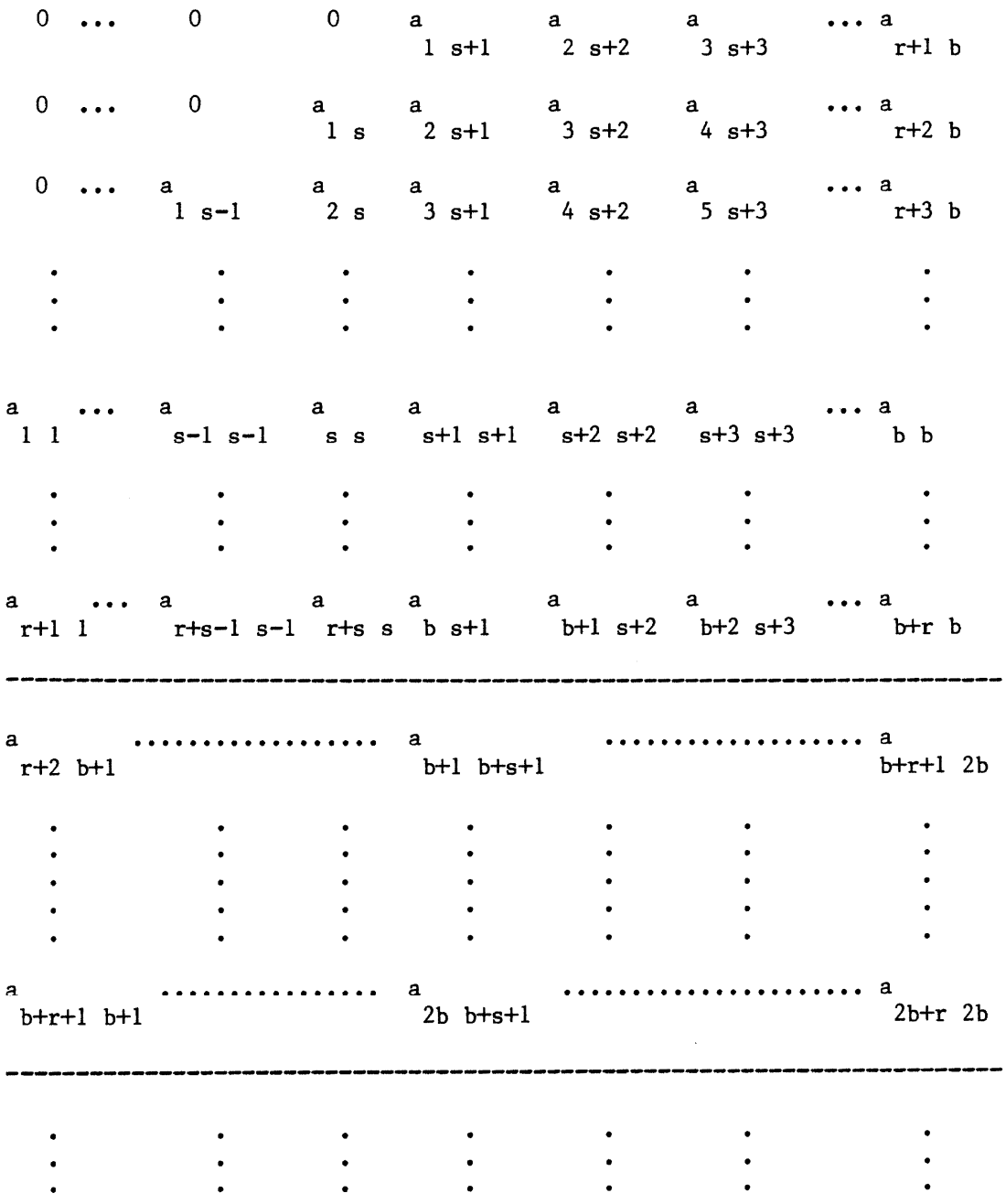

Figure 12: The matrix A formed into the data streams of 3C

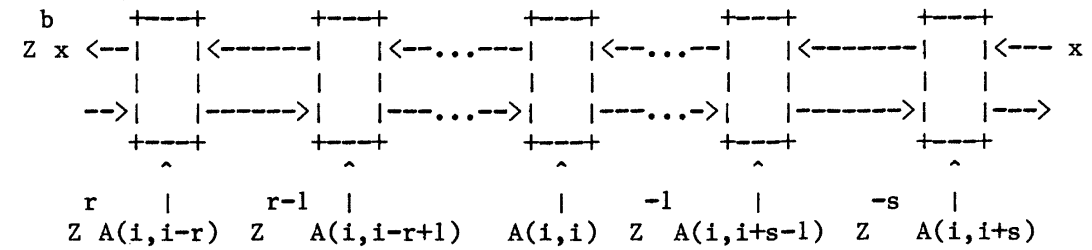

Figure 13: A codiagonally oriented array for matrix-vector products

The vector x is passed from right to left at a pace of one component for every other time step in a synchronous mode of operation. Each processor receives successive elements of one codiagonal. The streams forming the codiagonals are skewed with respect to each other. The set of data entering the array concurrently is indicated by asterisks in Figure 15. A processor unit is shown in Figure 14.

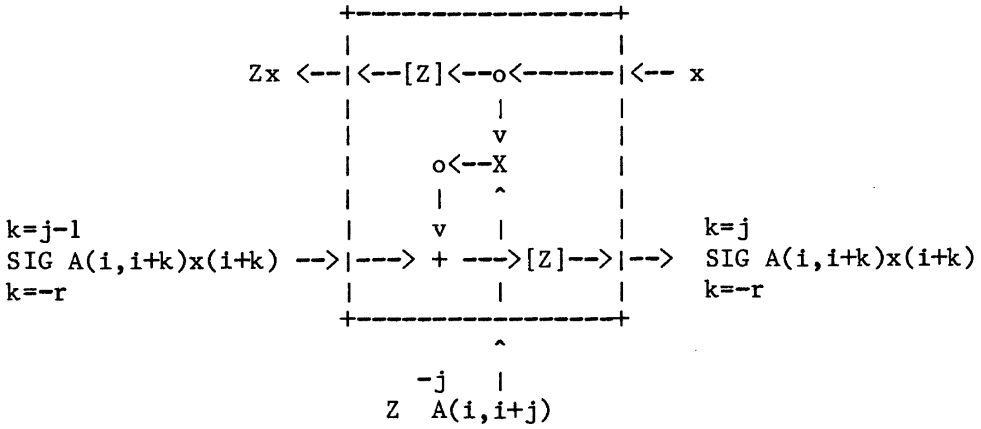

Figure 14: A processor in Figure 13

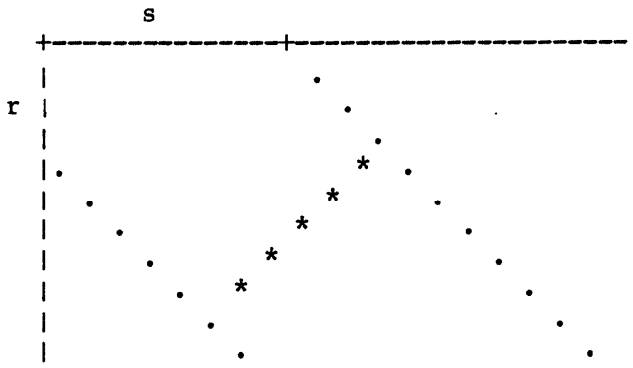

Figure 15: Wave fronts of the matrix-vector computation in Figure 13

Data in each data stream have to be separated by one time step. Hence, disregarding edge effects 2N time steps are required for the computation of the matrix-vector product.

Remark: There is only one wave front at a time moving one antidiagonal per time step. Since there are 2N antidiagonals it is easily seen that 2N time steps are required.

E) The reason for the processors being used only 50% of the time in D is that the data streams for the vector and the partial product sums are moving in opposite directions.

In the following algorithm the data streams have the same direction. Each processor works on a codiagonal. The product of a vector component and a column of the matrix is computed over successive time steps by a succession of processors. Figure 16 shows graphically the functions to be performed by a processor. Index j refers to the processor number where the ordering is in the direction of the flow starting with j=0 for the first processor.

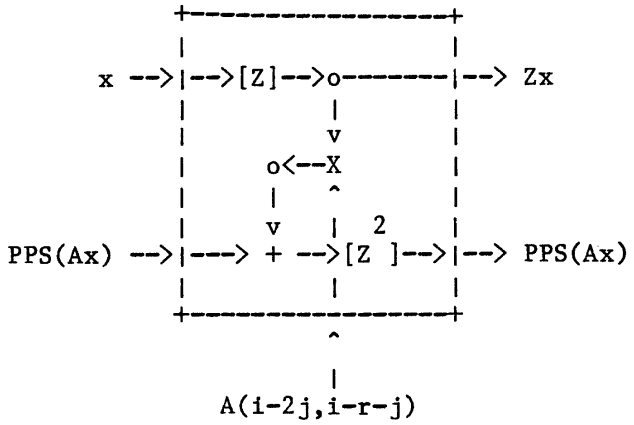

Figure 16: A codiagonal processor for matrix-vector products

The wave fronts illustrating the set of concurrent computations in an array using processors as in Figure 16 is as shown in Figure 17 where identical numbers denotes matrix elements that are entering the array during the same time step. The location of the numbers corresponds to elements of the matrix.

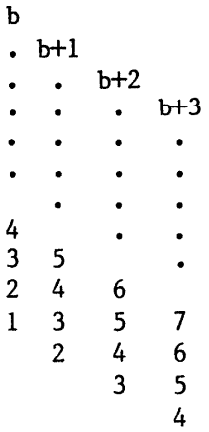

Figure 17: Wave fronts for an array based on codiagonal processors

F) Several more alternatives can be invented. For instance, successive processors in an array can compute successive odd or even components of Ap. In so doing the computations start at successive instances of time in neighboring processors. After b time steps a processor can start the computations of a new component of Ap at a distance of b from the previous one. Hence, the 2b processors compute the product in time N+b.

Remarks:

- The wave fronts of alternative 3D coincides with the data streams of alternative 2C, and the data streams of 3D with the wave fronts of alternative 2C. In a sense, these two alternatives are each other's duals.

- Alternatives 3A and 3B corresponds to N by b matrices obtained by shifting sets of b columns k*b columns, where k is an integer. Alternative 3C is obtained by shifting sets of b rows k*b rows. For alternative 3C two alternatives were discussed, one with the wave fronts forming codiagonals, one with wave fronts forming antidiagonals. Since the matrix is either N by b or b by N the compute time is N+b.

- Alternatives 3D and 3E can be thought of as being obtained after rotation of the matrix 45 degrees. In both alternatives codiagonals forms input data streams. However, the wave fronts in 3D falls on antidiagonals and the compute time is 2N, while the wave fronts in 3E are not antidiagonals and propagates faster. Another way of thinking of alternative 3E is as if the input was a b by N matrix obtained by shifting column i+1 with respect to column i up by one row. The wave fronts of alternative 3E, will in this representation, be codiagonals.

2.1.2 Computation of (x,Ax)

Using alternative 1 all components of the vector Ax are available during the same time step. Using a tree the product (x,Ax) can be computed in $\log_2 N$ time by 2N-1 processors. The combined algorithm requires 2Nb-1 processors and computes (x,Ax) in $\log_2 Nb$ time.

All versions of alternative 2 delivers all components of the product Ax during the same time step. As with alternative 1, a tree can be used to compute (x,Ax) in $\log_2 N$ steps for alternatives 2A and 2B and $\log_2 N+1$ steps for alternative 2C. The total number of processors for computing (x,Ax) based on 2A and 2B is 2N-1, and for 2C 4N-1.

All versions of alternatives 3 compute the product Ax in O(N) time. Hence, it is of interest in this case to consider O(N) alternatives for

computing (x,Ax), in particular if the computation of the inner product can be performed concurrently with computing Ax.

Alternative 3A delivers b components of Ap during every bth time step. A tree of 2b-1 processors can be used to form the inner product of vectors of dimension b in $\log_2 b$ steps. For each set of b components except the last the computations of (x,Ax) can be performed concurrently with computation of Ax. A total of 3b-1 processors will compute the inner product in $N+b+\log_2 b$ steps. With 2b-1 processors an additional N/b steps are required.

The b outputs from alternative 3A can also be loaded into a parallel load shift register. Using one processing element the product (x,Ax) can be formed in N+2b steps using b+1 processors.

Alternative 3B and 3C allows for the computation of (x,Ax) in N+s steps using b+1 processors.

Alternative 3D [Kung, Leiserson 80] requires 2N time steps and b+1 procesors to compute (x,Ax).

The alternatives for computation of (x,Ax) is summarized below:

Alternative	Processors	Intermediate storage	Time
1	2Nb-1		$\log_2 Nb$
2A, 2B	2N-1		$b+\log_2 N$
2C	4N-1		$b+1+\log_2 N$
3A, version 1	3b-1		$N+b+\log_2 b$
3A, version 2	2b-1		$N+N/b+b+\log_2 b$
3A, version 3	b+1	b	N+2b
3B, 3C	b+1		N+s
3D	b+1		2N

Table 1: Number of processors and time complexity for computing (x,Ax)

No distinction has been made between processors that according to the algorithms only would require an adder and those that requires additional facilities. The reason for this uniform treatment is that in most cases an implementation would use processors that are identical or close to identical for all processing needs.

In the table above versions 1 - 3 of alternative 3A refers to different ways of computing the inner product.

Alternative 3A, version 2 is in general of no interest since it in most cases is inferior to version 3. Version 3 does not offer any particular advantage over alternatives 3B and 3C.

Alternative 2C uses twice as many processing elements as alternatives 2A and 2B to compute the product in the same number of steps. The functionality of each processing element is somewhat simpler in

alternative 2C than in the other two cases, but would not make the processor half the size of the processors of 2A and 2B were they directly implemented in silicon. Alternatives 2A and 2B are also superior from a fault tolerance point of view.

Alternative 3D use twice the time of alternatives 3B and 3C. The processors of alternative 3D could be made somewhat smaller than those of alternatives 3B and 3C were they directly implemented in silicon. Alternatives 3B and 3C are superior from the point of fault tolerance.

In the following only alternatives 1, 2A, 2B, 3A (version 1), 3B and 3C will be considered.

3 CONCURRENT ALGORITHMS FOR THE CONJUGATE GRADIENT METHOD

The major computational task in each step of the CGM is the computation of the inner product (p,Ap). For the computation of the coefficients a_i and b_i a few more inner products are required. These inner products can be computed concurrently with (p,Ap) given that the processing capability is available.

3.1 Alternative 1

A concurrent algorithm for the CGM based on alternative 1 can compute a_i in one more time step than is given in table 1 by computing the products of the inner product (r_i,r_i) in the same processors as are computing the products of (p,Ap). The r_i*r_i terms can be passed towards the root ahead of the terms of (p,Ap). The value of a_i can be sent down the tree $\log_2 N$ levels and x_{i+1} and r_{i+1} computed at that level. For the next iteration it is also necessary to update the direction vector p_i, which requires the computation of b_i. If b_i were calculated directly from the definition the 2-norm of r_{i+1} would be needed. In another $\log_2 N$ steps the norm could be calculated. These time steps can, however, be avoided by rewriting the equation for b_i.

Substituting the equation for r_{i+1} into the expression for b_i and observing the property that $(r_i,r_j) = 0$, $i \neq j$ the following equation can be derived

$$b_i = -1 + a_i(Ap_i,Ap_i)/(p_i,Ap_i)$$

The values of a_i and (p_i,Ap_i) are already available in the root of the tree. The value of (Ap_i,Ap_i) can be available in the root one time step after (p_i,Ap_i) by computing the products of the inner product after those of (p_i,Ap_i) and passing the terms towards the root. Hence, the coefficients b_i can be computed in the root one or two time steps after a_i has been computed.

The components of the new direction vector p_{i+1} can be computed

concurrently at the leaf level in one step.

The total time for a step of the CGM is $2\log_2(Nb) + k$ (k=2-3).

3.2 Alternative 2

A concurrent CGM algorithm based on alternative 2 is the following:

Compute Ax_1, then r_1 and the products of (r_1,r_1) in the leaves of the tree. Pass the products towards the root to form the 2-norm. Concurrently compute Ap_1 in the leaves, then the products of (p_1,Ap_1) and pass the products towards the root to complete the inner product. After the products of (p_1,Ap_1) are computed compute the products of (Ap_1,Ap_1) which also are passed towards the root. Receive a_1 at the leaf level $2\log_2N$ time steps after the products of (p_1,Ap_1) was sent to the root and compute x_2. Compute r_2 during the following time step. Receive b_1 and compute p_2. What remains to be computed before one step of the CGM is completed are the products of (r_2,r_2) which can be performed concurrently with the computations of Ap_2 if the capability is available, otherwise another time step is required.

The total time for a step of this algorithm is $b + 2\log_2N + k$(k=3-4).

The leaf cells are active during b+k time steps out of $b + 2\log_2N + k$ time steps for one step of the CGM. The rest of the tree is used for summation during three time steps for every step of the CGM and for broadcasting during two time steps. The root performs arithmetic operations during four time steps.

It should be noticed that the topology of this algorithm corresponds to a ring of processors where the processors also forms the leaves of a binary tree.

3.3 Alternative 3

A binary tree of 2b-1 processors can be used to compute the inner products of alternative 3 in the previous section. The coefficients a_i and b_i can both be computed within two time steps after the computation of (p_i,Ap_i). Sets of b components of x_{i+1}, r_{i+1} and p_{i+1} can be computed on receipt of a_i and b_i at the leaves of the tree of height $\log_2b$. After the computation of the first set of b components of p_{i+1} the computations of Ap_{i+1} can start.

In alternative 3A b-r components of p_{i+1} are needed initially and one new component of p_{i+1} every time step thereafter. If there is a facility for computing b components of p and shifting them r processors, then no new components will be required in the computation during the r first time steps. However, if the components of p are calculated "where needed" a new component of p is required immediately. Two time steps

are required to compute a new set of b components of p since b components of r also have to be computed. Also, within the b time steps needed for the computation of b components of Ap the computations of the squares of the b components of r have to be carried out.

Hence, the time for a step of the CGM is $N+a*N/b+b+2\log_2 b+k(k=3-4)$, where a=0 for $b \geq 5$ and otherwise a=5-b.

The 2-norm of r_i can be computed concurrently with (p_i, Ap_i) by one additional processor to the b+1 processors of alternatives 3B and 3C. Using one additional processor (Ap_i, Ap_i) can also be computed concurrently with the other inner products. One additional time step is required for the computation of a_i and yet another for the computation of b_i. One additional processor is required for the computation of x_{i+1}, one for r_{i+1} and yet another one for p_{i+1}. These computations can be performed concurrently with the other computations of a step of the CGM. Hence, with this algorithm for a step of the CGM b+6 processors will perform the computations of a step of the CGM in N+s+4 time steps for alternatives 3B and 3C.

In summary the concurrent algorithms for the CGM have the following properties:

Alternative	Processors	Time/step
1	2Nb	$2\log_2(Nb)+k$
2	2N	$2\log_2 N+b+k$
3A, version 1	3b	$N+b+2\log_2 b+(a*N/b)+k$
3A, version 3	b+6	N+2b+k
3B,3C	b+6	N+s+k

Table 2: Processors and time complexity for one step of the CGM

The constant k is in all cases less than 5.

Alternative 1 requires the fewest time steps, has the largest processor count and the lowest processor utilization.

Alternative 2 uses fewer processors than alternative 1, has higher processor utilization, but requires $b-2\log_2 b$ more time steps for each step of the CGM.

Alternatives 3B and 3C have the smallest processor*time product. Alternative 3A, version 1 is inferior to alternatives 3B and 3C both in terms of processor utilization and number of time steps. For large bandwidths the gain in performance compared to alternative 3A, version 3 may be significant. Hence, a processor per row with broadcasting of the vector p in computing Ap, or a processor per column, are often the most attractive alternatives.

4 MAPPING ONTO BOOLEAN N-CUBES

Boolean n-cubes are interesting buildable topologies for concurrent machines, [DeBenedictis 82], [Lang 82]. In this section we will comment on how the algorithms described previously can be mapped onto Boolean n-cubes.

The Boolean n-cube is a Hamiltonian graph. Hence a linear array or a ring can be imbedded in an Boolean n-cube with the same unit communication distance between processors, given that the number of processors are the same. Hence, the algorithms described under alternative 3 would map fairly directly onto an n-cube with the same number of processors without changing the performance with more than a few time steps. (The topology of alternative 3 is not exactly isomorphic to a ring).

A binary tree can be imbedded in a Boolean n-cube having the same number of processors as the number of leaf processors in the tree, without loss of performance if there is only one wave front in the tree. The imbedded tree is neither binary nor regular but the height is the same as the binary tree. In alternative 2 the leaves are active during one of the cycles that activity takes place in the rest of the tree. Hence, the number of time steps for a Boolean n-cube version of alternative 2 is at least 1/iteration more than stated for the ring-tree version. Indeed, one of the processors will be performing $\log_2 N$ additions. A Boolean n-cube version of alternative 2 would perform the computations associated with one iteration in $3\log_2 N + b + k$ time using N processors.

A similar result holds for a Boolean n-cube implementation of alternative 1.

5 PARTIAL INSTANTIATION IN SPACE

5.1 The number of processors matches some characteristic dimension of the problem

Alternative 1 can be thought of as an algorithm fully instantiated in space, while alternatives 2 and 3 represent various forms of folding the in space fully instantiated algorithm so that it fits on a network of a smaller number of interconnected processors.

The topology of the three basic alternatives is different. Alternative 1 is a binary tree, alternative 2 can be thought of as a tree with communication paths added between the leaves, and alternative 3 consists of (essentially) linear arrays.

As the number of processing elements are reduced the storage per processor has to increase, since the information to be stored is fairly

independent of the number of processing elements. In alternative 2 above, storage corresponding to the part of a row that falls within the band is associated with each leaf processor. The storage can be implemented as a circular shift register, i.e. the access is sequential and repetitive.

In alternative 3A and 3B N/b sections of rows of A are associated with each processing element. The sections are that part of the row that falls within the band. For alternative 3C the same applies to columns instead of rows. The algorithms 3A - 3C can be viewed as block algorithms with the computations on successive blocks being pipelined. As for alternative 2 the storage associated with each processor is functionally equivalent to a circular shift register. For alternatives 3A - 3C the length of each of them is N/b.

A tree can be mapped onto a smaller tree in several different ways. One way is to fold the tree so that for a balanced tree of even degree two distinct nodes at the same distance from the root and joined by a path through the root in the original tree are merged into one node in the new tree. For instance, the root of a binary tree has two descendants, each being the root of a subtree. Applying the principle of folding described here, the two subtrees will be mapped onto each other to form one subtree in the obvious way. To form a new tree the root of the old tree can be mapped onto the root of the subtree.

Another way in which the number of processors can be reduced to half for a binary tree is if each processor assumes the tasks of its children. The tree is effectively shortened one level.

In both the above cases of reducing the size of a tree the mapping can be applied recursively. If a tree of n leaf nodes are maped onto a tree of m leaf nodes, $m<n$, then, assuming one task per processor in the larger tree, the following holds:

# leaves	#tasks in root	# tasks in other nodes
n	1	1
m	2n/m-1	n/m

If the subtrees of height $\log_2 b$ in alternative 1 are reduced to one node $2N(b-1)$ processors are saved at the expense of an increase in the computing time of $b-\log_2 b$. Each processing element in the reduced tree needs to store b components of the vector in the matrix-vector product. Alternative 2 that has the same number of processing elements and the same time complexity for computing the product stores each component of the vector only once.

Hence, the additional communication paths provided by alternative 2 compared to the above reduced version of alternative 1 saves a total of $2N(b-1)$ storage locations. In the CGM a new vector is computed for every step of the iteration process. In the reduced alternative 1 each leaf processor has to compute b components requiring b time steps. In

alternative 2 one step suffices. Duplication of information storage can be avoided at the expense of additional communication and increased time complexity from $2(\log_2 N+b)+k$ to $2(b+1)\log_2 N+k$.

Alternative 1 can be further reduced to a tree of 2b-1 processors to make it have a processor complexity comparable to alternative 3. The time complexity will then be $2(N+\log_2 b)+k$. With b-1 processors the time complexity will be $4N+2\log_2 b+k$.

5.2 An arbitrary number of processing elements

5.2.1 Band matrices

For the cases where the number of processing elements are O(N) and O(b) the characteristic dimensions of the problem can be used advantageously. The control and data management is simple. The mapping becomes more complex when the number of processors, m, is different from N and b.

The following are a few different ways of using m processors to compute the matrix-vector product. The extension of the algorithms to form a complete step of the conjugate gradient algorithm is straight forward.

- a) Broadcasting of the vector for a matrix-vector product.

  The number of processing elements m < b

  The m processors receives one row each. The vector p is broadcasted to the m processors. If the rows are consecutive the computations of the product Ap effectively starts and ends during consecutive cycles. Hence the time required for the computation of m components of Ap is b+m-1, if the components of Ap are computed in order of increasing component index. By taking sets of m rows skipping b-m rows in every path through the matrix the time for each set of m rows is reduced to b. The set of matrix coefficients supplied to the processors can be illustrated as follows when the sets of m rows are contiguous.

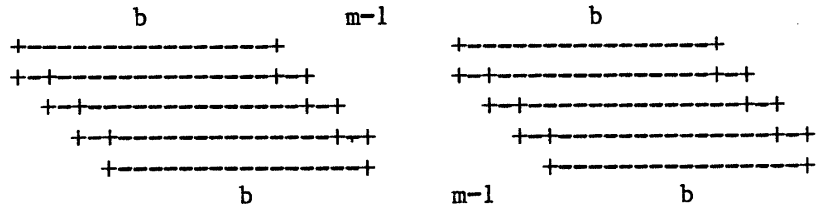

Figure 18: One possible sequence of row coefficients of A to a set of m processors

Some of the vector components have to be furnished more than once. In the case of contiguous rows the last b-m components

of p in every set of b components have to be supplied also for the computation of the next set of m components of Ap. Each processor performs a multiply-accumulate operation with a reset after every b operations. A simple shift register for p is not very suitable since after every b shifts forward b-m shifts backwards are required. Skipping b-m rows allows the components of p to be fed continuously. The starting component depends on which pass through the matrix is being made.

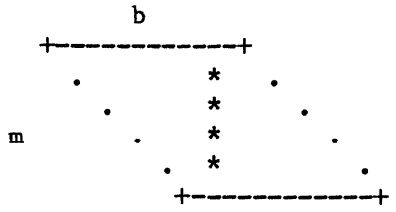

Figure 19: Wave fronts with broadcast input to m processors

The number of processing elements m > b

One way of considering this case is to use b processors to compute part of Ap as previously discussed and use the remaining m-b processors to compute the last few components of Ap in the manner described for m<b. This use effectively means that the broadcasting is limited to sets of b or fewer processors. (Note: At most b processors (rows) use the same component p.)

b) No broadcast. The components of p are being shifted into the processing elements as in Figure 1. The characteristics of this form of implementation are similar to a), the difference being that the m components are available simultaneously.

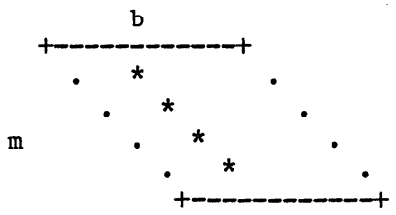

Figure 20: Wave fronts with sequential input to m processors

c) Instead of an algorithm based on rowwise scanning of the matrix it is obviously also possible to have the processors compute the matrix-vector product columnwise. When m<b partial product sums have to be stored and entered into the processor that is first in the direction of flow of the partial product sums. As for alternative 3C, the partial

product sums can flow from left to right if column i precedes column i+1 by 1 time step and from right to left if column i+1 precedes column i by one time step.

d) An algorithm can also be based on alternative E, which computes the product of a submatrix of A consisting of m adjacent codiagonals and a vector. By properly entering partial products to the first processor the product Ap can be obtained after b/m passes, m<b, each requiring N+b time steps.

Using the algorithm of [Kung, Leiserson 80] each pass requires 2N+b time steps and forms a partial product of m codiagonals. The total time for computing Ap is b/m(2N+b), i.e., twice the number of steps required by an array based on E.

For m>b the computations in c) and d) can be split so that different sets of processors compute different components of Ap reducing the time required for each pass.

5.2.2 Full matrices

The above algorithms for band matrices can also be extended to full matrices. For algorithms based on column or row ordering of data, the length of a major computation cycle as defined by the bandwidth is implemented as a control function in time. It should be noticed that for a full matrix the number of columns or rows are N, while the number of codiagonals and antidiagonals are 2N. Hence, for a given number of processors only half as many sweeps are required for column or row oriented algorithms as for algorithms based on codiagonals or antidiagonals.

The column or row sweep algorithms can be thought of as block algorithms where the blocks are N by m or m by N matrices. Other partitionings are obviously possible, but offer no particular advantage from a concurrency point of wiev.

6 SUMMARY AND CONCLUSIONS

Complexity: The conjugate gradient algorithm has global communication in each step of the solution procedure. The global communication is inherent in the inner product (p,Ap). Measured in time where one unit of time is the time required for performing one multiplication/addition and one communication, at least $O(\log_2 N)$ units of time is required for the matrix-vector inner product, even if all the multiplications are performed concurrently. For a linear system of equations the solution is computed in N steps by the conjugate gradient method, where N is the

dimension of the problem. The devised algorithms have a time complexity ranging from $N\log_2 N$ to N^2, and use a number of processors ranging from $2N^2$ to N.

Algorithms devised for band matrices have a lower time or space complexity. The number of processing elements required for a direct implementation of the algorithms devised here range from 2Nb-1 to b+a, ($a \leq 6$) and the time complexity from $2\log_2 Nb+k$ to N+s+k, $k<5$.

Concurrent algorithms suitable for machines with an arbitrary number of processing elements, i.e., a number of processing elements that are not directly related to any characteristic dimension of the problem, are also described.

All the algorithms discussed have the characteristic that one communication is associated with each computation. In some algorithms only input data is subject to a communication action for each computation. In others, partial results are also communicated for each computation.

Sparsity: Most of the algorithms are devised for band matrices. The algorithms also apply to full matrices with the proper change in either the number of processing elements or in the control or both. Algorithms based on row or column organization of the data may be preferable to algorithms based on co- or antidiagonals since there are N of the first kind and 2N of the diagonals.

Sparsity within the band is disregarded in the above algorithms. In alternative 1 accounting for the sparsity would imply a certain pruning of the tree. In alternative 2A sparsity can be exploited to reduce storage requirements in each processor at the cost of additional logic. Due to the interlocking of different row computations the time would still be b time steps. For 2B the interlocking is caused by partial product sums. The time is again b, and exploiting sparsity would only reduce storage.

In alternatives 3, the computation time is determined by the size N of the vector and the matrix. Hence, exploiting sparsity of A would at first affect storage and processor requirements. As in alternative 2 reducing storage requirement by exploiting sparsity is fairly straightforward, but the interlocking caused by passing of vector components or partial product sums does not allow for any reduction in the number of processing elements nor reduction in time complexity without a significant change in the algorithms.

Fault-tolerance: In alternative 1 the failure of any leaf processor causes one component of the matrix-vector product to be erroneous. The error stems from the faulty product. A failure of any of the processors in the subtrees of height $\log_2 b$ will cause a partial product sum to be in error, and hence the corresponding component to be in error. A similar argument applies to the next $\log_2 N$ levels of the tree with respect to the computation of the innerproduct (p,Ap).

In alternative 2A the failure of any arithmetic unit only affects one

component of the matrix-vector product. In alternative 2B b components are affected by a fault in one arithmetic unit, while in 2C b/2 components are affected.

In alternatives 3A and 3B one out of every b rows are affected by an error in any arithmetic unit. In alternative 3C a fault in one arithmetic unit implies that all matrix-vector components are erroneous, as is the case for alternatives 3D and 3E.

REFERENCES

[DeBenedictis 82]
DeBenedictis, E., Seitz, C.L.
Homogeneous Machine - Technical Plan.
Technical Report 4705:DF:82, California Institute of Technology, 1982.

[Glowinski et. al. 80]
Glowinski, R., Lions, Mantel, B., Periaux, J., Pironneau, O., Poirier, G.
An Efficient Preconditionned Conjugate Gradient Method Applied to Nonlinear Problems in Fluid Dynamics via Least Square Formulations.
North-Holland, 1980, pages 445-487.

[Hestenes, Stiefel 52]
Hestenes, M.R., Stiefel E.
Methods of Conjugate Gradient for Solution of Linear Systems.
J. Res. Nat. Bur. Standards 49:409-436, 1952.

[Jennigs, Malik 78]
Jennings, A., Malik, G.M.
The Solution of Sparse Linear Equations by the Conjugate Gradient Method.
Int. J. Numer. Methods 12:141-158, 1978.

[Kershaw 78]
Kershaw, D.S.
The Incomplete Cholesky Conjugate Gradient Method for the Iterative Solution of Systems of Linear Equations.
J. Comput. Phys. 26:43-65, 1978.

[Kung, Leiserson 80]
Kung, H.T. and Leiserson, Charles E.
Algorithms for VLSI Processor Arrays.
In Introduction to VLSI Systems, pages 271-294. Addison-Wesley, 1980.
Mead, Carver A. and Conway, Lynn A.

[Lam 76]
Lam, B.
Methods for Solving Large Sparse Indefinite Systems of Linear Equations.
Technical Report TR No. 53, The Computer Centre, The Australian National University, July, 1976.

[Lang 82]
Lang, C. R.
The Extension of Object-Oriented Languages to a Homogeneous, Concurrent Architecture.
Technical Report 5014:TR:82, California Institute of Technology, 1982.

[Manteuffel 80]
Manteuffel, T.A.
Solving Structures Problems Iteratively with a Shifted Incomplete Cholesky Preconditioning.
North-Holland, 1980, pages 427-444.

[Meijerink, van der Vorst 77]
Meijerink, J.A., van der Vorst.
An Iterative Solution Method for Linear Systems of which the Coefficient Matrix is a Symmetric M-Matrix.
Math. Comp. 31:148-162, 1977.

[Meijerink, van der Vorst 78]
Meijerink, J.A., van der Vorst.
Guide lines for the useage of incomplete decompositions in solving sets of linear equations a occur in practical problems.
Technical Report TR-9, ACCU, Utrecht, 1978.

[Munksgaard 79]
Munksgaard, N.
Solving Sparse Symmetric Sets of Linear Equations by Preconditioned Conjugate Gradients.
Technical Report Report CSS 67, Harwell, 1979.

[Saad 80]
Saad, Y.
The Lanczos Biorthogonalization Algorithm and Other Oblique Projection Methods for Solving Large Unsymmetric Systems.
Technical Report UIUCDS-R-80-1036, Dept. Computer Science, UNiversity of Illinois, Urbana-Champaign, December, 1980.

[Saad 81]
Saad, Y.
Krylov Subspace Methods for Solving Large Unsymmetric Linear Systems.
Technical Report UIUCDCS-R-81-1047, Dept. Computer Science, University of Illinois, Urbana-Champaign, January, 1981.

[van der Vorst 82]
van der Vorst, H. A.
A Vectorizable Variant of Some ICCG Methods.
SIAM, Scientific and Statistical Computing 3(3):350-357, 1982.